Prettyprinting in an Interactive Programming Environment

Martin Mikelsons
Computer Science Department
IBM T. J. Watson Research Center
Yorktown Heights, New York

ABSTRACT: Prettyprint algorithms designed for printing programs on paper are not appropriate in an interactive environment where the interface to the user is a CRT screen. We describe a data representation and an algorithm that allow the efficient generation of program displays from a parsed internal representation of a program. The displays show the structure of the program by consistent and automatic indentation. They show the program in varying levels of detail by replacing unimportant parts with ellipsis marks. The relative importance of program parts is determined jointly by the structure of the program and by the current focus of attention of the programmer.

INTRODUCTION

Prettyprint algorithms designed for printing programs on paper are not appropriate in an interactive environment where the interface to the user is a CRT screen. A carefully formatted listing is a very readable representation of a program. However, when this listing is presented on a screen, each instantaneous view presents only a fragment of the program and most of the time important contextual information is not visible. In an interactive programming environment [1, 2, 4, 5] where the system has access to the parsed representation of the program, and where the system has a record of the recent editing activity of the user, we can profitably go beyond conventional prettyprinting methods to present a useful and coherent view of the program to the user at the display console.

We can determine what part of the program is currently of interest to the user. We can elide portions of the program that seem irrelevant to the current picture in order to bring closer together widely separated but important features. We can suppress the text that represents the detailed structure of statements and expressions in order to show an overview of the structure of a program. We can use abbreviations and special symbols to condense the text. We can use different fonts or colors to distinguish keywords from identifiers. In short, each instantaneous view of the program can be generated to show the features and components that are significant at that particular moment. Figure 1 shows for a simple example the difference between our approach and conventional prettyprinting.

The notion of a condensed form for expressions has been a feature of Lisp expression editors from their earliest origins [3, 6]. More recently, several interactive programming environments support some form of ellipsis [2, 4, 5]. In all cases, however, the level of detail is controlled by a user parameter or by explicit suppression of specific phrases. We feel that a programming tool should not require that degree of user involvement in the ubiquitous process of filling the screen after each interaction.

(a) A window on a file

```
            call b;
      end;
    else
      do;
        x=x+1;
        y=y+1;
      end;
  end;
```

(b) A window on a program

```
a:PROC(x,y);
    ...
    IF x<y
       THEN ...
       ELSE DO; x=x+1;
                y=y+1;
            END;
  END;
```

Figure 1. Two views of a program fragment

```
IF f(x, y, z)
     THEN DO; x=1;
              y=2;
              z=3;
             END;
     ELSE DO; x=2;
              y=3;
              z=4;
             END;
```

Figure 2. A sample compound statement

Overview of the Display Generation Process

There are many parameters that control the format and content of the kind of display we have described. We could require that the user specify all these parameters. But since we are dealing with an interactive programming environment, the user is constantly issuing commands that manipulate and modify the program shown on the screen. We can usually infer from these commands what part of the program is of interest to the user and how it should be displayed. We have isolated three major modes that cover most of the activities that programmers engage in when editing programs. Each mode defines a focus of attention and a strategy for displaying the contents and context of the focus.

Edit Mode: In edit mode, the user is examining or changing a program in detail. The focus of attention is taken to be the most recently selected or modified phrase; the strategy is to show some of the context in which the focus occurs, and an adequate amount of detail within the focus. The relevant context is usually some indication of the phrase that contains the focus, such as a DO group or IF statement, and some of the phrases that immediately precede and follow the focus. Figure 3 shows several views of a statement in edit mode. Edit mode is also in effect during interactive debugging of a user program. In this case, the focus is selected by the interpreter to show the progress of execution through the program.

Multiple Focus Mode: In multiple focus mode, the focus of attention is a set of phrases selected from the program by a search or an analysis function of the system. The display strategy is to show all or most of the phrases in the focus and some of the program structure that contains them and that connects them together. Figure 4 illustrates this mode of display. This mode can also be used to bring together on one screen two or more widely separated parts of a program in order to compare them.

Reading Mode: This mode is used to read a program sequentially. In reading mode, the strategy is to show a maximum of detail in the center of the display area and to force most ellipses to the first and last lines; by providing commands that shift this window to the next part of the program, we simulate in a more structured way the process of scrolling through a source file. Figure 5 shows two successive reading mode screens.

(a) From the top:

```
IF f(x, y, z)
     THEN DO; x=1;
              y=2; ...
             END;
     ELSE DO; x=2; ...
             END;
```

(b) From inside the THEN clause:

```
IF f(x, y, z)
     THEN DO; x=1;
              y=2;
              z=3;
             END;
     ELSE DO; x=2; ... END;
```

(c) From inside the ELSE clause:

```
IF f(x, y, z)
     THEN DO; x=1; ... END;
     ELSE DO; x=2;
              y=3;
              z=4;
             END;
```

(d) After adding a statement to the ELSE clause:

```
IF f(...) THEN DO; ... END;
     ELSE DO; x=2;
              y=3;
              z=4;
              w=5;
             END;
```

Figure 3. Edit Mode: Four views of the statement in Figure 2 on six 30 character lines.

Once we have determined what part of the program is of interest to the user, our display generation has two main steps. First, we generate a data structure that contains the information necessary to format each part of the program. Second, given a specific screen area, a focus of attention, and a strategy that determines the relative importance of program units, we allocate the available display area to a subset of the program text.

```
IF f(x, y, z)
     THEN DO; x=1;
              y=2; ...
             END;
     ELSE DO; ... y=3; ...
             END;
```

Figure 4. Multiple Focus Mode: The statement in Figure 2 on six 30 character lines after selecting all occurrences of y in the statement.

```
IF f(x, y, z)
     THEN DO; x=1;
              y=2;
              z=3;
             END;
     ELSE DO; x=2; ... END;
```

```
IF f(x, y, z) THEN ...
     ELSE DO; x=2;
              y=3;
              z=4;
             END;
...
```

Figure 5. Reading Mode: Two views of the statement in Figure 2 on six 30 character lines.

The first step is to generate from the concrete parse tree of the program a tree structure of objects we call boxes. While the parse tree reflects the syntactic distinctions determined by a formal grammar, the box structure reflects the semantic and esthetic distinctions involved in formatting a program.

One function of the box structure is to simplify the decisions made during formatting. While the grammar of a typical language may define hundreds of non-terminal symbols, or node types, the box structure is composed of fewer than ten different kinds of boxes. The box types reflect formatting concepts such as functional grouping, concatenation and vertical alignment.

A second, equally important function of the box structure is to allow re-structuring of the parse tree. For example, many grammars parse lists of statements into left or right recursive binary nodes. The natural representation for formatting purposes is an ordered list of statements. Another example of re-structuring is a subscripted variable such as 'a(i)'. A natural decomposition would be into two parts 'a' and '(i)', but many grammars parse this example into a structure where the subscript part does not exist as a node.

The box structure must be generated from a top-down scan of the concrete parse tree in order to recognize correctly the various constructs of the language.

The second step is the display allocation process. In contrast to the top-down nature of box generation, this step is typically an inside-out process. The key parameters to the display allocation process are a window that defines the available display area, a focus that defines the user's focus of attention in terms of the box structure, and a strategy that defines the relative priorities of the actions possible during the process. The goal of display allocation is to fill the window with text that shows the context and contents of the focus. The resource to be allocated is the available display area; the contenders for this resource are the parts of the program within and around the focus. Since allocation is done with respect to the box structure, it is entirely independent of the source language.

In order to create a display we simply send to the display device the characters allocated by the second step. This is the only device-dependent part of the process.

In the following sections, we first describe the structure of boxes and how they are used to represent the semantic and esthetic decisions involved in formatting a program. We then describe the allocation algorithm and how it is varied to reflect the strategies necessary to implement the three main display modes. Since all the steps in display generation may be invoked as a result of each user interaction with the system, the necessary computations must be performed within severe time constraints. We show how the algorithm can be tuned to meet these constraints. In the last section we describe how box generation is parametrized with respect to language.

BOX CHARACTERISTICS

The purpose of each box is to summarize the formatting information necessary to display that particular part of the program. The purpose of the box structure is to represent the nesting structure of the components of a program. There are atomic boxes that describe individual symbols in a program, and composite boxes that describe groups of symbols and boxes. Each box also contains a number of flags and parameters that define the desired format for that box. A key feature of the box structure is that it defines at each point in the program a collection of possible displays that consume increasing amounts of display area. In all cases the parameters represent intentions, and not exact specifications.

Properties of Program Text

A salient property of program text is the fact that a program reads from left to right as a one-dimensional string of symbols; at the same time, the meaning of the program is defined by a tree structure of nested composite objects. The program string is made readable by breaking it up into symbols and groups of symbols and by separating these groups on different lines of a page. Membership in a higher grouping is shown by vertical alignment while depth of nesting is shown by the degree of indentation. We note however that when a structure is small, it is not necessary to lay it out vertically to make it readable; the eye can parse it as well as the machine.

In an interactive display situation we can use these properties to advantage in order to increase the amount of useful information on each screen. Small structures can be shown on several lines if they are near the focus of attention; they can be shown condensed on one line if they are part of relatively less important context. Large structures may also show as small structures if they are condensed by ellipses.

Figure 6 and Figure 7 show some of the box structure for the statement in Figure 2.

Atomic Boxes

Atomic boxes describe an occurrence of a symbol in a program. The main information associated with an atomic box is the text of the symbol. The text in an atomic box is identified as either a special symbol, a keyword, an identifier or a constant.

Keywords and identifiers are often not distinguishable in the source string, yet in the abstract representation, their meaning is very different. By distinguishing keywords from identifiers in the box structure, we can identify them in the display. The simple expedient of showing keywords in uppercase characters and identifiers in lowercase characters enhances the readability of a program. We can also distinguish them by color or font on a display device with that capability.

Constants can be shown as entered, in canonical form, or in abbreviated form, to achieve different effects in the display.

Composite Boxes

Composite boxes define the components of a composite object in a program. The primary attribute of a composite box is a list of component boxes. In addition, composite boxes contain attributes that define how the components should be formatted relative to each other.

The vertical positioning of text in a composite box is determined by explicit or implicit cut points. If a box is a potential cut point, then the text within that box may begin on a new line of the display. If a box is not a potential cut point, then the text in that box is always concatenated to the text on the current line. In a vertical box, every immediate component is an implicit cut point. In addition, the components of a vertical box must all appear on the same line, or they must all appear on distinct lines. In horizontal boxes there are no implicit cut points; any potential cut points in a horizontal box must be explicitly declared during box generation. In a block box, every component is also an implicit cut point, but several components may appear on a line.

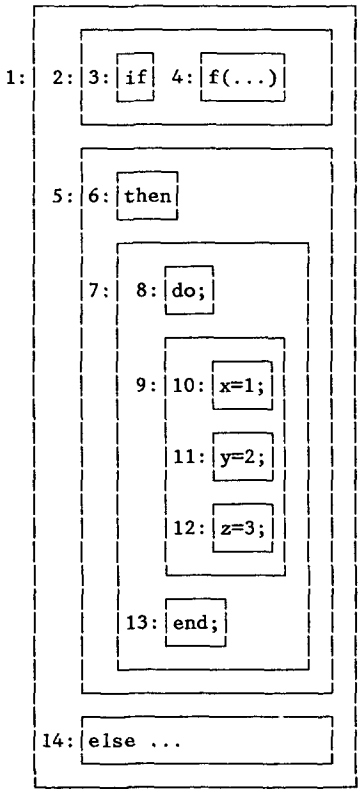

Figure 6. Part of the box structure for the statement in Figure 2

Box	Kind	Components	Other attributes
1	Horizontal	2, 5, 14	indent=5
2	Horizontal	3, 4	required expansion
3	Atomic		required expansion keyword
4	Block		
5	Horizontal	6, 7	cut point
6	Atomic		required expansion keyword
7	Horizontal	8, 9, 13	indent=4
8	Block		required expansion no separating spaces
9	Vertical	10, 11, 12	conditional cut point merge inner ellipses separate parts with space
10, 11, 12	As required		
13	Block		required expansion offset=-2
14	Similar to 5.		

Figure 7. Details of the boxes in Figure 6

A composite box must specify the indentation of the components relative to the origin of the box. We also want to specify whether the components must be separated by a space if they appear on the same line, and whether adjacent elided components can be merged into a single ellipsis. If the standard priority allocation is not suitable to the components of a box we can specify the relative priorities of the components.

Relation to Nearby Boxes

Each box has attributes that specify the relationship of that box to its display environment. In addition to the implicit spacing of components, we may want to specify the leading and trailing spaces required for a box. This parameter is applied only if the box is adjacent to another on the same line. We can also specify an offset that shifts the origin of the box relative to the current left margin. Since this number may be positive or negative, it can be used to shift text to the left where it stands out in a sequence of indented line.

Each box can have an explicit cut point declaration. High priority cut points are allocated when encountered; low priority cut points are allocated only if the line containing them overflows the display area. Conditional cut points are allocated only if the origin of the box overlaps text on the current line.

Miscellaneous Flags and Parameters

A box can be flagged as a potential focus or a potential frame. A potential focus is a box that contains a phrase of the abstract syntax. A potential frame is a box that may be the outermost box that shows in a display window. In a PL/1 DO-group, for example, the group may be formatted as a box of three components: the header, the body, and the ending statement. All three components may be selected as a focus, but it may be desirable to specify that the body box is not suitable as a frame.

A box can be marked as requiring expansion. In this case whenever an attempt is made to show the box at all, an attempt must be made to show the components instead. This feature prevents important connective keywords or delimiters from being hidden in ellipses.

Display features such as brightening, font or color can be invoked by specifying a static or dynamic highlight class. The static highlight class of a box is determined during box generation, while the dynamic highlight class changes during a session. The distinction between keywords and identifiers is an example of static highlight class. The focus is normally identified in every display by marking the appropriate boxes with a dynamic highlight class.

Static and dynamic highlight class are used to index into a table of effective highlight classes which are in turn mapped into an effect on the display. On a display device with several colors, several effective highlight classes can be displayed simultaneously. On a display device with only one highlight possibility, such as brightening, only one effective highlight class can be shown at one time, and the meaning of the displayed highlight must be identified in an accompanying legend. But the box structures are identical for the two devices.

Special Purpose Text

Comments in most programming languages require special handling. Since comments are normally removed from the text by the lexical scanner, they do not even appear in the parse tree. It is also desirable to show a program with or without comments, or to replace sections of code by an associated comment as a form of user controlled ellipsis.

A programming environment generates a large amount of descriptive information about the program being edited. A natural way of displaying this information is in the form of footnotes or asides in the display of the program.

Since the box structure does not have to be identical to the parse tree, we can include comments and annotations in the places that are most natural for them. By identifying comments and annotations in the box structure, we can generate displays with or without them by simply changing a parameter to the display allocation algorithm.

THE DISPLAY ALLOCATION ALGORITHM

The goal of the display allocation algorithm is to select a subtree, W, from the box structure, B, that represents the entire program. The root of W need not be the root of the program, and the leaves of W need not be the leaves of the program. Where the leaves of W coincide with leaves of B, the display shows the full text of the program; where the leaves of W are a sub-tree of B, the display shows an ellipsis.

The subtree W is generated as a sequence W(0), W(1), ... of subtrees where each W(i) is a proper subtree of W(i+1). Given that the display of W(i) fits in the available screen area, the potential successors of W(i) are defined either by an ascent step or by one of several expansion steps. An ascent step includes W(i) in a larger subtree; an expansion step extends a leaf of W(i) to include more of the detailed structure of B. W(i+1) is the first potential successor of W(i) that fits on the screen; if there are no potential successors, or none of them fit, then W is W(i). The ordering of the potential successors is determined by relative priorities assigned to the ascent step and to the possible expansion steps.

A possible implementation of the allocation algorithm could be to start at F and assign priorities to all the boxes in B. We could then examine the boxes in B in order of decreasing priority and

perform ascent or expansion steps as necessary until W was found. This implementation is slow because it must repeatedly scan portions of B that will never appear on the screen.

First, we describe a faster algorithm that combines the assignment of priorities with ascent and expansion steps and that implements this process by a marking process on B. The potential successors and their priorities are generated by each ascent or expansions step. We then describe refinements that increase the efficiency of the process, and that improve the appearance of the resulting display. We will describe the process for edit mode and then indicate how it is modified to implement multiple focus mode and reading mode.

Data Structures and Initialization

The marking of B is done with two numbers a, and c. The number a marks boxes that are definitely part of W; the number c is used to mark boxes in a potential successor. If the potential successor fails, the boxes in it remain marked with an obsolete value of c and are not scanned again, otherwise they are re-marked with a. The number c is incremented by one for each potential successor attempt. The numbers are initialized to x+1, where x is the highest value of c in the preceding display computation. As a result, the marking performed during one display computation cannot be confused with markings left over from previous passes.

The data objects used by the algorithm are a priority queue P, and three lists R, S, and T. P contains boxes and ascent marks ordered by priority. An ascent mark is a dummy box that is used as a place holder with a specific priority. R is a list of boxes that start a new line in a potential successor; S is a list of new boxes included in a potential successor; T is a list of boxes at the fringe of S.

The initial contents of the queue P is determined by the current focus and the current display mode. In edit mode, with a single focus, we set the priority of the focus to 1 and enter the focus in P. If the focus is not the root of P, we also add an ascent mark to P.

The Basic Algorithm

The display allocation process consists of three procedures described here in an informal programming language. The main entry to the algorithm is the procedure **allocate**. The procedure **expand** describes an expansion step, and the procedure **ascend** an ascent step.

```
allocate: procedure;
 top: If empty_queue(P) then return;
      b := first_in_queue(P);
      R, S, T := empty_list;
      c := c+1;
      If is_ascent_mark(b)
           then call ascend;
           else call expand(b);
      call measure;
      If ¬window_overflow
           then begin;
                  add_list_to_queue(T, P);
                  for y in R call mark(y, a);
                  for y in S call mark(y, a);
                end;
           else continue;
      go to top;
    end allocate;

expand: procedure(x);
      call mark(x, c);
      call add_to_list(x, S);
      If is_atomic_box(x) then return;
 /* Otherwise  x  must be composite. */
      For each y in box_components(x)
           if has_req_expand_flag(y)
                then call expand(y);
                else call add_to_list(y, T);
      return;
    end expand;

ascend: procedure;
 top: x := box_ancestor(W);
      call mark(x, c);
      For each y in box_components(x)
           if y=W then continue;
           else if has_req_expand_flag(y)
                    then call expand(y);
      else call add_to_list(y, T);
      If ¬has_pot_frame_flag(x)
           then go to top;
      If has_ancestor(x)
           then call add_ascent_mark;
      Return;
     end ascend;
```

The purpose of the procedure **measure** is to determine whether a potential successor fits or can be made to fit in the available display area. The measuring process is a depth first scan of B, starting at the root of W and including all boxes marked with the current values of a and c. Lines are allocated on the basis of allocated cut points.

As text producing boxes are encountered in the scan, the effective length of the current line is incremented appropriately. When an allocated cut point is encountered, a new line is started with an initial indent determined by the indent parameter of the ancestor of the cut point and the offset parameter of the cut point. Cut points, like expansions and ascents, are marked with the numbers a or c.

During this scan, we compute the length of each line of the display at the current level of expansion, and we build a priority queue D that contains potential cut points ordered by priority. If all the lines are within the limits of the display area, we discard D and return to the main algorithm. If there are lines that overflow the display area and if there are lines still

available, we remove entries from D , mark them with c as allocated cut points, and place them in R. At this point we must repeat the measuring process.

Line Allocation

The process of line allocation interacts closely with the state of the display at every stage of expansion. If lines are allocated too slowly, expansion steps will fail early and the resulting display will not fill all the available space. If lines are allocated too readily, then the resulting display will appear thin and will not make good use of the available horizontal space.

As each line is generated during the measuring scan, required cut points are entered in D as they are encountered. Optional cut points are remembered until the end of the line is reached; if the line fits in the available space, the optional cut points are ignored, otherwise, they are added to D with a degraded priority. If the text on a line is inside a block format box, the only possible cut point that is recognized is the first component of the block that begins within the available space and ends outside of it.

Priority Allocation

The priority of boxes during the expansion process is determined by a number that is computed for each box as it is reached. Increasing values of this number indicate progressively lower priority. In Edit Mode, when a composite box x with priority p is expanded, the priorities of the components are up, up+v, up+2v, ... , where u and v are constant parameters of the algorithm. When an ascent step is made from a box x with priority p, the priority of the ancestor of x is sp, the priorities of the components to the right of x are p+t, p+3t, ... , and the priorities of components to the left of x are p+2t, p+4t, ...

By varying the parameters s, t, u and v, we can vary the appearance of the display in terms of the amount of context or detail that shows relative to the focus. At the same time we retain a consistent appearance that favors the focus and looses detail as distance from the focus increases.

The relative priority of potential cut points is determined by the structure of B. Thus when the focus is on B as a whole, the cut point priorities are the same as the expansion priorities. However, when the focus is on a subtree of B, the cut point priorities retain their top-down bias and the shape of the display in terms of vertical layout and indentation remains roughly constant as the focus shifts.

Efficiency

We can see from the above description that if k is the number of symbols showing in a final display, and n is the average number of components in each box, then in the worst case we will iterate nk times through the allocation algorithm. Since each iteration invokes a measuring step that will visit an average of k boxes, the total number of boxes visited to compute a display is nk^2. We can reduce the number of times the measuring step is invoked, and we can reduce the number of boxes visited during each measuring step. Note that in any case, the cost of computing a display is controlled by the size of the screen, not by the size or complexity of the program being displayed.

The expansion process can be divided into two successive stages. The first stage lasts until all the available lines have been allocated. During this stage the measuring pass must record potential cut points. As soon as all the lines have been allocated, we enter the second stage in which the measuring process is much simpler and faster. Since most of the calls to the measuring procedure are in the second stage this optimization is very effective.

It is not necessary to measure the entire display after each expansion step. If the expansion affects only one line, does not cause an overflow, and does not introduce any cut point candidates, then we can confirm the expansion and continue without re-measuring. We cannot compute the exact effect of the expansion from purely local information but we can estimate the maximum effect that could be produced.

During the first stage of expansion, this is the best we can do, but in the second stage of expansion, we can make a much more accurate estimate. For each line we maintain a minimum possible length and a maximum length. For each expansion, we make a minimal and a maximal estimate of the effect of the expansion. If the minimal estimate causes the minimal length to overflow, then the expansion step will definitely fail in measuring, so we can discard the expansion without measuring. The maximal estimate works just as in the first stage.

We can reduce the number of boxes scanned during the measuring process in the second stage by detecting boxes that have reached a stable form. An atomic box is stable if it was fully expanded in a previous step, i.e. it is marked with a; any box is stable if is permanently elided, i.e. it was abandoned in a previous step and is marked with a value between a and the current value of c. Furthermore, a box is stable if all its components are stable. During each measuring step, we can propagate summary information to the ancestors of stable boxes and skip that descent subsequently.

Refinements

The allocation algorithm as described will show short phrases with low priority next to ellipses that stand for long phrases of higher priority. For example, the phrase

```
big_function_name(big_var_name+x)
```

may show as

```
...(...+x)
```

if only 10 character positions are available for it. This turns out to be unesthetic. We can remedy this situation by removing from P any immediate siblings of a box that fails the expansion step.

In order to implement the intention of vertical boxes, we must introduce a cleanup step at the transition from stage one to stage two. For each line that contains more than one but not all the components of a vertical box, we suppress the expansion of all but the highest priority component on that line.

In order to implement the intention of block boxes, we must allow the text in a block to flow from one line to the next as inner boxes are expanded. This means that we must re-allocate the cut points in a block each time the block is measured. In order to do this, and still be able to discard an allocation simply by stepping the value of c, the cut point allocation in blocks must contain a memory of the previous allocation.

The allocation algorithm described above can generate a display with multiple foci if we change the initial value of P and introduce an additional stage of expansion that precedes the two previously mentioned stages. Let X be the set of desired foci. We initialize P with the elements of X and compute Y, the least common ancestor of the boxes in X. We then run the allocation algorithm with the following modifications. For each box b removed from P, if b is already expanded, we ignore it; otherwise we perform an ascent step. When W contains Y, we switch to the normal algorithm.

In reading mode, the focus and priority strategy are determined by the scrolling direction desired. If the user scrolls towards the end of the program, the new focus is the first elided phrase to the right of the current focus. The priority strategy is to assign increasing numbers from the new focus in a depth-first, left-to-right manner. If the user scrolls toward the head of the program, the new focus is the first elided phrase to the left of the current focus. The priority strategy is to assign increasing number in a depth-first, right-to-left manner.

Display Generation

The generation of the actual display buffer is a special case of the measuring step. Instead of computing the effective size of a box, we lay down the text in a display buffer. This step is clearly device dependent. With some devices, such as the IBM 3277, the device dependency must be carried back into the measuring process also since brightening may introduce a space on the display where the box structure did not require one.

In many cases however, a change in dynamic and effective highlight class can be effected by regenerating the display from the most recent allocation. As a result, shifting brightening from one allocated phrase to another, can be done very quickly.

BOX GENERATION

We have described the box structure and the display allocation process in terms of a data structure distinct and separate from the parse tree. Clearly there are many points at which there is a one-to-one correspondence between parse tree nodes and boxes. Many of the differences between the parse tree and the box structure arise from the fact that all nodes in the parse tree have a bounded number of descendants, while boxes do not. Other differences are caused by arbitrary choices made in the grammar of the language. It is not clear whether we can design a grammar for a given language in such a way that it allows efficient parsing and at the same time reflects the intuitive abstract structure of the language.

In the current implementation, the box structure is generated from the parse tree as a separate data structure by a collection of routines that embody the esthetic choices involved in prettyprinting as lines of code. A more general approach would be to define box generation as a collection of pattern-constructor pairs where the pattern matches a fragment of the parse tree and the constructor defines a box structure.

If the parse tree is allowed to change, as it must be in a program development environment, we must update the box structure accordingly. The simplest approach is to recompute the entire box structure whenever any change is made to the parse tree. A much more effective solution is to propagate change markers from the point in the parse tree where a change is made to the root of the parse tree. Then, during the box generation step if all the nodes involved in a prettyprinting decision are unmarked nodes, we can use the previously generated box structure for that portion of the parse tree. This method is very natural if user updates to the program are incorporated into the existing parse tree by an incremental parser [7].

Syntax errors, comments and annotations cause similar problems during box generation. Since they can occur at any point in the program, it is impractical to account for them in the patterns that drive the prettyprinter. If we add them by updating the box structure generated from the parse tree, then we are updating a structure generated under the assumption that it specifies a complete format. The current implementation appends all comments and errors to the atomic box that immediately precedes them. This has often unfortunate effects on the format of the display. We hope to achieve a more pleasant format by treating unparsed text in a manner similar to the parsed text around it. Thus a comment between two statements would be treated like a statement.

CONCLUSION

The algorithms described in this paper have been implemented in Lisp, for a prototype of a programming environment that supports an extended subset of PL/1. We are currently re-implementing the prototype in its target language. The efficiency of

the Lisp implementation is adequate for demonstration purposes. If a user interaction does not cause a display allocation pass, such a shifting the focus to a phrase on the current screen, the display can be updated in about 50ms. of cpu time. If the display allocation procedure is invoked, the required cpu time is from 300 to 500 milliseconds. In the first case, the response is instantaneous in our time-sharing environment; in the second case there is a definite delay in system response. By using more compact and efficient data structures, we expect to speed up the response of the new implementation by a factor of 5. This should provide instantaneous response in all cases and allow the system to be used as a serious programming tool.

This method of generating displays is not limited to programs. We have used it for data structures and in a very limited way for text. Clearly, any structured object that cannot be shown fully on a single screen can be condensed with appropriate ellipses to make each view of it more readable.

In conclusion, our experience shows that this method of generating displays of programs works, and is adequately efficient. An important effect of this approach is that it emphasizes the point of view that a program is an abstract object that is defined by the structure of its components, in contrast to the view that the program is a slab of text from which the structure is inferred every time it is read.

ACKNOWLEDGMENTS

The idea of program text expanding to fill an available window originated in early discussions of program editors with Gerry Howe and Vincent Kruskal. I would also like to thank my colleagues Cyril Alberga, Steve Fortune, Paul Kosinsky, George Leeman and Mark Wegman for many arguments and criticisms that helped form the ideas in this paper.

REFERENCES

1. Alberga, C. N., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M., and Wegman, M. N., A Program Development Tool, Eighth Annual ACM Symposium on Principles of Programming Languages, January 1981, 92-104.

2. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B. and Levy, J. J., A Structure-oriented Program Editor: A First Step Towards Computer Assisted Programming, International Computing Symposium 1975, North Holland Publishing Company, 1975.

3. Hawkinson, L., and Kameny, S. L., LISP Edit Program, LISPED, System Development Corporation, TM-2337/100/01, April 1966.

4. Swinehart, D. C., COPILOT: A Multiple Process Approach to Interactive Programming Systems, Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford University, July 1974.

5. Teitelbaum, R. T., The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS, Department of Computer Science, Cornell University, TR 79-370, June 1979.

6. Teitelman, W., INTERLISP Reference Manual, Xerox Palo Alto Research Center, 1976.

7. Wegman, M. N., Parsing for Structural Editors, Twenty-first Annual IEEE Symposium on Foundations of Computer Science, October 1980, 320-327.